Express Mail Label No. EV 318163308 US

[0001] OPERATING SYSTEM (OS) ABSTRACTION LAYER

[0002] CROSS REFERENCE TO RELATED APPLICATION(S)

[0003] This application claims priority from U.S. provisional application no. 60/405,995, filed August 26, 2002, which is incorporated by reference as if fully set forth.

[0004] FIELD OF INVENTION

[0005] This invention generally relates to wireless devices. In particular, the invention relates to developing software to run on hardware of such devices.

[0006] BACKGROUND

[0007] Developing software to effectively use hardware, such as communication processors and system processors, of wireless devices is difficult. Wireless devices have limited memory storage capacity, making it desirable that software operating systems, interfaces and applications have a small footprint. However, using reduced footprint software design techniques limits the flexibility accorded to software designers.

[0008] A difficulty in wireless device software design is the application programmer's interfaces (APIs). Currently, several alternative open platform operating systems (OSs) are available for wireless devices, such as Symbian EPOC, Microsoft Windows CE and Palm OS, and several alternative real time operating systems (RTOS) are available, such as OSE, DSP BIOS and an alternate RISC RTOS. It is desirable to have an API that is capable of targeting a protocol stack, such as a third generation partnership project (3GPP) universal mobile telecommunication system (UMTS) protocol stack, to all the different combinations of RTOSs and OSs.

[0009] SUMMARY

[0010] An operating environment is capable of being abstracted to a plurality of operating systems. An operating environment is provided which is common to all the different operating systems. A plurality of operating system abstraction layers are provided. Each abstraction layer designed to abstract the operating environment to at least one targeted operating system.

[0011] BRIEF DESCRIPTION OF THE DRAWING(S)

[0012] Figure 1 is a simplified diagram of one configuration of a wireless device with software and programmers interface illustrated.

[0013] Figure 2 is an illustration of messaging between SDL processes using light integration across partitions.

[0014] Figure 3 is an illustration of messaging between SDL processes within a partition using tight integration.

[0015] Figure 4 is an illustration of messaging between SDL processes across partitions using tight integration.

[0016] Figure 5 is a flow chart for processing using thread groups.

[0017] Figure 6 is a high-level illustration of the OS abstraction layer.

[0018] Figure 7 is an illustration the OS API software structures.

[0019] Figure 8 illustrates the design of the message queue construct.

[0020] Figure 9 illustrates the message queue control block data structures.

[0021] Figure 10 is an illustration of an object database implemented in an OS-independent manner.

[0022] Figure 11 is an illustration of a time manager object.

[0023] Figure 12 is an illustration of time manager control block functionality.

[0024] Figure 13 is a flow diagram of a creation/initialization time manager scenario.

[0025] Figure 14 is a flow diagram of a get OS time manager scenario.

[0026] Figure 15 is a flow diagram of a set CallBack time manager scenario.

[0027] Figure 16 is a flow diagram of a cancel CallBack time manager scenario.

[0028] Figure 17 is a flow diagram of an execute CallBack time manager scenario.

[0029] Figure 18 is an illustration of the message queue with IPC elements.

[0030] Figure 19 is a flow diagram of message queue creation and destruction in context with inter-processor communication.

[0031] Figure 20 is a flow diagram of message queue opening in context with inter-processor communication.

[0032] Figure 21 is a flow diagram of message queue remote put in context with inter-processor communication.

[0033] Figure 22 is an illustration of a message QID database.

[0034] DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

[0035] Figure 1 is a simplified diagram of one configuration of a wireless device with software and programmers interface also illustrated. One such wireless device is a wireless transmit/receive unit. A wireless transmit/receive unit (WTRU) includes but is not limited to a user equipment, mobile station, fixed or mobile subscriber unit, pager, or any other type of device capable of operating in a wireless environment. Although the preferred embodiment uses the hardware configuration of Figure 1, alternate hardware configurations can be used with the wireless device software.

[0036] The wireless device has a system processor 100, such as a RISC. The system processor 100 typically runs the system applications, such as web browsing, calendars, etc. A communication processor 102, such as a DSP, typically runs the wireless communication protocols. A communication module 104 allows for communication between the system and communication processor. A shared memory 106 is accessible by both processors 100 and 102 via the communication module 104.

[0037] Figure 1 also illustrates the preferred software structure of the wireless device and its association with the hardware. The SDL model layer 108 represents the communication protocol stack components at very high level of abstraction. The SDL model environment 108 is composed of SDL processes 118_1 to 118_6 (118) communicating at a peer-to-peer level via SDL signals. SDL model elements (processes and signals) are transformed into operating environment elements by targeting the code generator to the operating system adaptive port interface (OS API) 120.

[0038] The SDL porting layer 110 represents the conversion of SDL model elements (processes, signals, partitions, etc.) into operating environment constructs through the OS API 120. The operating environment 112 provides an OS/hardware independent environment via the OS API 120. The preferred operating environment 112 operates independent of the underlying OSs. As a result, the software is independent of the operating system. Accordingly, if the underlying OSs change, the software does not have to be rewritten. The OS abstraction layer 114 implements the OS API 120 for a particular underlying operating system and hardware platform. It maps the OS API 120 constructs into their underlying OS representations. The abstraction layer 122 provides a direct implementation for functionality that an underlying OS does not support. For example, if an OS does not support messages and message queues, the OS abstraction layer would provide its own implementation.

[0039] The OS abstraction layer interfaces with the communication module 104 to support inter-processor communication. The preferred OS abstraction layer 122 allows for device drivers to cross processor boundaries. To illustrate, an audio driver may need to be present for the system OS 124, but the audio hardware may be present on the DSP 102. The communication module 104 provides the communication path between the OS API threads 128_1 to 128_2 (128) across the system/communication processor boundary and is responsible for the transfer for both data and control. As shown in Figure 1, a queue 130_1 to 130_2 (130) is associated with each thread 128. The communication module 104 provides the communication path between OS API threads across the system/communication processor boundary. It is responsible for the transfer of both data and control. A system OS 124 is associated with the system processor 100 and the communication OS 126 is associated with the communication processor 102.

[0040] Although, all of the layers work together in the preferred software structure, each layer can be typically used with other layers in alternate software structures.

[0041] Figures 2 to 4 illustrate of communication between SDL processes 118. Figure 2 is an illustration of messaging between SDL processes 118 using light integration. SDL processes 118 execute on top of an SDL Kernel (K/E) 132_1 to 132_2 (132). The SDL kernel 132 manages communication between the processes 118, routing signals to their appropriate recipients. The SDL model 108 may be divided into separate light integration partitions. Each partition contains its own SDL kernel 132, and resolves to a single thread 128 with a queue 120 within the operating environment 112. Each partition's SDL kernel 132 manages communication between internal SDL processes 118 without OS assistance. The SDL kernel 132 forwards signals destine for SDL processes 118external to its partition through the operating environment 112via messages 134_1 to 134_2 (134).

[0042] Initially, the signal from one SDL process 118_2 is converted to an OS message by the SDL kernel 132_1 in a first environment. The message is routed to an SDL kernel 132_2 in a second environment. In other words, the message is "put" in the second environments queue 130_2. The SDL kernel132_2 in the second environment detects the message, and converts the received message back to its signal format. The second environment SDL kernel 132_2 signals the appropriate process.

[0043] Fundamentally, each SDL light integration partition resolves to a thread with a single message queue within the operating environment. Communication between partitions occurs via messaging. Each SDL kernel / environment knows what SDL processes 118 are internal, and what SDL processes 118 are external. Each SDL kernel / environment knows where to route external signals (i.e. what partition their recipient resides) and each SDL kernel / environment knows how to convert signals to messages, and vice versa.

[0044] Figure 3 is an illustration of messaging between SDL processes 118 within a partition using tight integration. Using tight integration, the SDL model 118 maps directly to the operating environment 112. All SDL process communication (signaling) takes place through the operating environment 112. The SDL kernel (SDL process scheduler) is not included using tight integration. Specifically, each SDL process resolves to a thread 128 with a single message queue 130 within the operating environment 112. Communication between SDL processes 118 takes place through the operating environment 112. The SDL porting layer 110 converts signals to messages 134, and vice versa.

[0045] For a first SDL process 118_1 to signal a second SDL process 118_2, the first process 118_1 converts the signal into an OS message. That message is routed to the second SDL process thread 128_2 via its message queue 130_2. That SDL Process thread 128_2 wakes up on the message and the message is converted back to a SDL signal. The second SDL Process 118_2 receives the signal from first SDL Process 118_1.

[0046] Figure 4 is an illustration of messaging between SDL processes across partitions using tight integration. For a first SDL process 118_1 to signal a second SDL process 118_2, the first process 118_1 converts the signal into an OS message. That message is routed to the first partition external task 136_1. This environment is the environment acting as a proxy for all processes external to this partition. That partition external task 136_1 forwards the message (signal) to the second partition External Task 136_2. The second partition external task 136_2 routes the message (signal) to the second process thread 128_2 via its message queue 130_2. That thread wakes up on the message. The message is converted back to a SDL signal and the second SDL process 118_2 receives the signal from the first SDL process 118_1.

[0047] This procedure facilitates talking across processor boundaries. The partitions can by targeted to a specific processor and to function across the boundaries external tasks 136 may be used.

[0048] Abbreviations

ADT - Abstract Data Type

API - Application Programmer's Interface (Client Interface in this context)

CPU – Hardware (HW) Processor: system RISC (such as an ARM) or DSP.

IPC - Inter-processor Communication

OS - Operating System

HAL - Hardware Abstraction Layer

HW - Hardware

MUX - Mutex

SoC - System-On-a-Chip

SW - Software

[0049] Terms and Definitions

Communications Processor - Embedded CPU (DSP) executing low-level communication protocol stack layers, algorithms, device drivers, etc.

Operating Environment - Abstract programming environment provided by the OS abstraction layer through the OS API.

Open Platform - Operating system with a published (or publicly available) API. Third party organizations develop user-level applications to the published API.

Proxy - Placeholder or surrogate for another object, controlling access to it. Used in the OS Abstraction to present constructs physically located on a remote processor.

System Processor - General-purpose CPU executing an "Open Platform" operating system or a real time operating system (RTOS) including user-level applications.

Underlying Operating System - A target OS on which the OS abstraction layer and the OS API are built.

[0050] The operating environment 112 preferably hides the underlying OS and HW structure from its clients via an OS abstraction layer 122. This includes dual processor HW architectures. The operating environment 112 via the OS API preferably provide services to the following clients: SDL processes 118 (protocol stack model) and device drivers and software components.

[0051] The OS constructs are defined in an OS independent manner. Table 1 provides definitions for each of the OS constructs supported by the preferred OS API 120.

OS Construct:	**Definition:**
Thread	Independent path of execution. Managed on a priority basis.
Process	Independent path of execution (thread) with its own protected address space. Note, a process may contain more than thread.
Thread Group	Grouping of threads, managed collectively to synchronize their execution. Each thread within the group contains an independent path of execution, scheduled on a priority basis. The thread group synchronizes the initialization of all threads within the group. Basic Functions: - Synchronize – Bring-up or start all threads within the group, synchronizing them to a common initialization point. Specifically, all threads initialize, and block. - Start – Allow all threads within the group to begin their "main" path of execution after initialization and synchronization.
Mutex	Thread synchronization element providing mutual exclusion to shared resources: memory regions/buffers, hardware devices, etc.

OS Construct:	**Definition:**
	Related terms: Semaphore, Critical Section, etc. Basic functions: - Lock – Block if the MUX is locked by another thread, otherwise acquire the MUX and continue. - Release – unlock or release the MUX.
Event	Thread synchronization element, allowing threads to coordinate execution. A thread may inform another thread of some condition by signaling an event. Basic functions: - Set – signals the event. - Wait – blocks until the event is signaled or set. Events allow one thread to "pass control" to another thread.
Message	Communication/synchronization element between threads. A message allows threads to exchange information through a common interface. Messages allow one thread to "pass control" and to "pass data" to another thread.
Message Queue	Manages message communication between threads. Includes transparent communication across the processor boundary. Basic functions: - FIFO behavior. - Put – places a message on the queue. - Put After– places a message on the queue after an indicated delay. - Wait – returns the message at the head of the queue. If no messages reside on the queue, block until one does. - Get – returns the message at the head of the queue, if available. Does not block on empty. Variations: • Priority messages. Messages ordered on the queue based on relative priority. Across the processor boundary: message data copied from the sender's memory space to the receiver's memory space.

OS Construct:	Definition:
	Within a processor: no data copy. Pointers cannot be included within a message due to processor boundary restrictions.
FIFO	Manages transfer for blocks of data between threads, including transparent data transfer across the processor boundary. Simpler than messages. Basic Functions: - Put: places a memory block on the FIFO. - Get: pulls a memory block from the FIFO. - Wait: depends on the FIFO until a memory block appears. Pulls and returns. Across the processor boundary: data copied from the sender's memory space to the receiver's memory space. (This will depended on the HW architecture.) Within a processor: no data copy.
Linked List	Generic list, providing a common implementation. Basic Functions: - Add: place elements in the list. - Get: read elements in the list. - Remove: pull elements from the list. - Iterate: provides all three services while iterating over the list.

Table 1

[0052] Other miscellaneous OS elements are defined per Table 2.

OS Construct:	Definition:
Memory Allocation/De-allocation:	Memory allocation/de-allocation management. Several variants: • General: allocation from a generic system heap. • Local: allocation from a locally defined heap. • Communication: allocation of communication buffers (i.e. memory buffers passed up and down the protocol stack). Such buffers may need to be shared across the system/communication processor boundary.

OS Construct:	Definition:
Delay (Sleep)	Allows a thread to pause its execution for a specified unit of time (milliseconds).

Table 2

[0053] The SDL Porting layer 110 takes the SDL model elements, such as processes 118, signals, partitions, etc., and converts them into operating environment constructs. The following is the preferred embodiment of the OS API 120.

[0054] The OS API 120 has API entry points. All API entry points are named using the following convention:

os_<action><construct>(...)

Prefix - All API entry points are preceded with the "os_" prefix for identification.

Action - <action> field indicates what operation will be performed on the construct.

Construct Name - <construct> field identifies the construct receiving the action, or operation.

[0055] The OS API data structures are as follows. All the constants, defined in the API, are preceded with the "OS_" prefix for identification. All the API data types are preceded with the "Os_" prefix for identification. Individual enumeration values are preceded with the "OS_" prefix.

[0056] All Operating Environment elements are identified and referred to by a unique handle (Os_Handle_T). When the construct is created or opened, its assigned a handle, and that handle is used whenever the element is utilized.

[0057] All OS API entry points, with a few exceptions, return the generic OS Return Type (Os_Rtn_E). All time-relative API entry points (for example, os_putAfterMsgQ) return an OS Cancel ID to the caller (Os_CancelId_T). The Cancel

ID may be used to cancel or terminate the time-relative operation, using an associated cancel entry point (for example, os_cancelPutAfterMsgQ).

[0058] Thread Priority Level (Os_ThreadPriority_E) represents the relative priority assigned to a thread, or a synchronized thread with a thread group. Preferred thread priority levels are per Table 3.

OS_THREAD_PRIORITY_LEVEL0	Highest priority level.
OS_THREAD_PRIORITY_LEVEL1	.
OS_THREAD_PRIORITY_LEVEL2	.
OS_THREAD_PRIORITY_LEVEL3	.
OS_THREAD_PRIORITY_LEVEL4	.
OS_THREAD_PRIORITY_LEVEL5	.
OS_THREAD_PRIORITY_LEVEL6	.
OS_THREAD_PRIORITY_LEVEL7	Lowest priority level.

Table 3

[0059] Different operating systems may handle scheduling of threads at the same priority level differently. For example, some round robin "run-able" threads at the same level on a time slice or quantum basis. Others allow "run-able" threads to individually run to completion (i.e. until they block). EPOC falls under the former and OSE the latter. Accordingly, threads assigned to the same priority level are checked prior to the assignment.

[0060] The thread function defines "client provided" entry points into a thread or a synchronized thread. All entry points must be a "C" function accepting a void * argument and returning void, as defined by: Os_ThreadFunction_T.

```
typedef void (Os_ThreadFunction_T)( void *arg_p )
```

[0061] The OS Time MS represents time in milliseconds. One count or tick is 1 ms.

```
typedef Uint32  Os_TimeMS_T
```

[0062] A Mutex is a fundamental thread synchronization element providing mutual exclusion to shared resources. Mutexs are not shared across processes, or processor boundaries.

[0063] The following are Mutex services. Creates create an operating environment Mutex, allocating all required underlying OS resources. The caller provides a reference to a handle. This call will populate the handle reference with a unique value, identifying the created Mutex.

Prototype:

```
Os_Rtn_E
os_createMux( Os_Handle_T *handle_p );
```

Parameters:

*handle_p	Mutex Handle, filled in by this call on successful creation

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0064] The destroy entry point destroys or deletes an operating environment Mutex, releasing all allocated underlying OS resources. After this call, the Mutex Handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyMux( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the MUX.

Returns:

OS_RTN_OK	Successful deletion.

OS_RTN_BAD_PARAM	Invalid Mutex handle.

[0065] The lock entry point locks or acquires an operating environment mutex. If the mutex is locked by another thread, this call will block until the locking thread releases the mutex. If multiple threads are blocked waiting on the mutex, the thread with the highest relative priority will gain access to the mutex. The priority inheritance is proven/verified within the underlying operating system.

Prototype:

```
Os_Rtn_E
os_lockMux( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the MUX.

Returns:

OS_RTN_OK	Successful, mutex is locked.
OS_RTN_BAD_PARAM	Invalid Mutex handle.

[0066] The release entry point releases an acquired or locked mutex. This call will unblock the highest priority thread, pending on the mutex. The priority inheritance is proven/verified within the underlying operating system.

Prototype:

```
Os_Rtn_E
os_releaseMux( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the MUX.

Returns:

OS_RTN_OK	Successful, the mutex was released.
OS_RTN_BAD_PARAM	Invalid Mutex handle.

[0067] An Event is a thread synchronization element, allowing a thread to signal another thread of some condition.

[0068] The following are event services. The create entry point creates an operating environment event, allocating all required underlying OS resources. The caller provides a reference to a handle. This call populates the handle reference with a unique value, identifying the created Event.

Prototype:

```
Os_Rtn_E
os_createEvent( Os_Handle_T *handle_p );
```

Parameters:

*handle_p	Event Handle, filled in by this call on successful creation

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0069] The destroy entry point destroys or deletes an operating environment event, releasing all allocated underlying OS resources. After this call, the event handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyEvent( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Event.

Returns:

OS_RTN_OK	Successful deletion.

OS_RTN_BAD_PARAM	Invalid Event handle.

[0070] The set entry point sets an operating environment event. This call will unblock all threads, pending on the event.

Prototype:

```
Os_Rtn_E
os_setEvent( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Event.

Returns:

OS_RTN_OK	Successful, the event is set.
OS_RTN_BAD_PARAM	Invalid Event handle.

[0071] The pend entry point blocks an operating environment event, until it is set or signaled. If the event is set, this call clears the event and returns immediately. Otherwise, it blocks until another thread sets the event. When set, the blocked thread wakes, the event is cleared, and control returned to the caller.

Prototype:

```
Os_Rtn_E
os_pendEvent( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Event.

Returns:

OS_RTN_OK	Successful, the event has been set, and is now clear.
OS_RTN_BAD_PARAM	Invalid Event handle.

[0072] The clear entry point clears an operating environment event. If the event is not set, this service becomes a NOP.

Prototype:

```
Os_Rtn_E
os_clearEvent( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Event.

Returns:

OS_RTN_OK	Successful, the event is now clear.
OS_RTN_BAD_PARAM	Invalid Event handle.

[0073] The check entry point checks the state of an operating environment event, set or clear.

Prototype:

```
Bool
os_checkEvent( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Event.

Returns:

TRUE	Event is set.
FALSE	Event is clear, or any error condition.

[0074] A thread represents a single independent path of execution within the operating environment 112 and, in turn, the underlying software system. All threads within the system are scheduled on a priority, time-sliced basis via the underlying operating system. Threads within a single process share a common linear address space.

[0075] The thread services are as follows. The create entry point creates an operating environment thread, allocating all required underlying OS resources. If the suspend flag is clear, the thread will be started immediately, and its "Main Loop" will be invoked by the underlying OS scheduler based on its priority. Otherwise, the thread is suspended, and will not become schedulable until it is explicitly resumed. The "Main Loop" represents the thread's main path of execution. This function should never return. The caller provides a reference to a handle. This call will populate the handle reference with a unique value, identifying the created Thread.

Prototype:

```
Os_Rtn_E
os_createThread(
    Os_Handle_T              *handle_p,
    Uint32                   stackSize,
    Os_ThreadPriority_E      priority,
    Os_ThreadFunction_T      *mainLoop_fp,
    Bool                     bSuspended,
    void                     *argument_p);
```

Parameters:

*handle_p	Thread Handle, filled in by this call on successful creation.
stackSize	Stack Size for the created Thread in bytes.
Priority	Priority for the created Thread.
*mainLoop_fp	Reference to the Thread's Main Loop.
bSuspended	Suspended Flag. TRUE == create Thread suspended.
*argument_p	Argument for the created Thread. Passed to the Thread's Main Loop.

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_BAD_PARAM	Bad input parameter detected.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0076] The destroy entry point destroys or deletes an operating environment thread, releasing all allocated underlying OS resources. This call does not free resources that the thread may have allocated internally during its execution. To prevent leaks, all such resources are de-allocated before the thread is destroyed. After this call, the thread handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyThread( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Thread.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid Thread handle.

[0077] The suspend entry point suspends an operating environment thread. The underlying OS will not schedule the thread until it is resumed.

Prototype:

```
Os_Rtn_E
os_suspendThread( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Thread.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PA RAM	Invalid Thread handle.

[0078] The resume entry point resumes the execution of a suspended operating environment thread. The underlying OS will schedule the thread based on its priority. Prototype:

```
Os_Rtn_E
os_suspendThread( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Thread.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid Thread handle.

[0079] A thread Group coordinates / synchronizes the bring-up or initialization of a group of threads. Figure 5 is a flow chart for processing using thread groups. The thread group provides services to:

- Synchronize a collection of threads (i.e. bring-up all threads in the group to a common initialization or rendezvous point), 494.
- Start the collection, simultaneously (i.e. start each thread's main execution path after synchronization), 496.

[0080] All threads within a group must be specified when the thread group is created.

[0081] Thread groups do not utilize standard threads. Instead, a client provides the group with a set of specialized threads, called "SyncThreads". A SyncThread expands the base functionality found within a standard thread, extending it to include two entry points: an "Init" function and a "Main Loop" function. When the group is synchronized, the "Init" function will be invoked for each SyncThread within the group, based on its priority. The "Init" function must perform all internal activities required to initialize its thread, and return. For example, create a message queue, allocate some scratch pad memory, initialize a driver or some hardware, etc. When the group is started, the "Main Loop" function will be invoked within each thread by the underlying OS based on its priority. The "Main Loop" represents the thread's main path of execution. This function should never return.

[0082] Similar to standard threads, all SyncThreads within a single process share a common linear address space.

[0083] When a Thread Group is created, the client provides it with a SyncThread data table containing all the information required to create and manage the threads. Each entry in the table contains the Os_SyncThreadData_S data type as per Table 4.

stackSize	Stack Size in bytes.
priority	Thread Priority
*init_fp	Reference to the "Init" function for this SyncThread.
*initArg_p	"Init" Argument. Passed to the SyncThread's "Init" function.
*loop_fp	Reference to the "Main Loop" function for this SyncThread.
*loopArg_p	"Main" Argument. Passed to the SyncThread's "Main Loop" function.

Table 4

[0084] The SyncThread data table is terminated with null entry indication:

OS_SYNC_THREAD_NULL.

[0085] This entry point creates an operating environment thread group, creating all the SyncThreads within the group and their underlying OS resources. The create operation does not start the threads. The creating client must explicitly start them via the synchronize and start operations.

[0086] The caller provides a reference to a handle. This call will populate the handle reference with a unique value, identifying the created thread group.

Prototype:

```
Os_Rtn_E
os_createThreadGroup( Os_Handle_T         *handle_p,
                      Os_SyncThreadData_S *syncTable_p );
```

Parameters:

*handle_p	Thread group handle, filled in by this call on successful creation
*syncTable_p	Pointer to the first entry in this thread group's SyncThread table. The provided table must be made up of continuous Os_SyncThreadData_S entries, terminated with an OS_SYNC_THREAD_NULL.

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_BAD_PARAM	Bad parameter detected within the SyncThread Table.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0087] The destroy entry point destroys or deletes an operating environment thread group; destroying all of the group's SyncThreads and releasing all group allocated underlying OS resources. This call does not free resources that may have been allocated internal to any of the SyncThreads. To prevent leaks, all such resources are de-allocated before the thread group is destroyed. After this call, the thread group handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyThreadGroup( Os_Handle_T handle );
```

Parameters:

handle	OS Handle, identifying the Thread Group.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid Thread Group handle.

[0088] The initialize / synchronize entry point synchronizes all SyncThreads within an operating environment thread group. It ensures that each SyncThread within the group has initialized. Specifically, this call will not return until all threads execute their "Init" function and return.

Prototype:

```
Os_Rtn_E
os_initThreadGroup( Os_Handle_T handle );
```

Parameters:

Handle	OS Handle, identifying the Thread Group.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid Thread Group handle.

[0089] The start entry point starts all SyncThreads within an operating environment thread group. This call may only be called after the group has been synchronized. This call allows each SyncThread within the group to enter its "Main Loop" function, which occurs based on the priority of each thread. This call does not wait for the SyncThreads to execute. It just makes them schedulable by the underlying OS, and returns.

Prototype:

```
Os_Rtn_E
os_startThreadGroup( Os_Handle_T handle );
```

Parameters:

handle	OS Handle, identifying the Thread Group.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid event handle.

[0090] A Message is a thread synchronization element, allowing a thread to signal another thread with both control and data. They must be used in tandem with message queues. Specifically, a thread can allocate a message, populate it with data, and transfer it to another thread by "putting" the message on a message queue. Messages may cross the processor boundary.

[0091] Messages may contain any combination of data types: by value for which the data is contained directly in the message and by reference for which pointers to data is in the message.

[0092] Pointers within messages are used carefully. Pointers will not translate across processor boundaries. Also, if a message must be destroyed internally within OS abstraction layer 122, the OS layer 122 cannot not free data blocks passed by reference within the deleted message. They must be managed externally by the client to prevent leaks. As such, passing data by reference (via pointer) within a message should be avoided.

[0093] The allocate entry point allocates a single message. It returns a void * pointer to the caller, referring to the memory block representing the message. The size of this block equals the required message size (plus a small message header, hidden from the client). After allocating a message, the client may populate the memory block with data.

Prototype:

```
Os_Rtn_E
os_allocMsg( Uint32    size,
         void    **msg_pp );
```

Parameters:

Size	Size to allocate for this message.
**msg_pp	Pointer to return a void * pointer, referring to the allocated message.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid size.
OS_RTN_MEM_ERROR	Insufficient memory available to satisfy this request.

[0094] The free entry point frees or de-allocates a single message.

Prototype:

```
Os_Rtn_E
os_freeMsg( void    *msg_p );
```

Parameters:

*msg_p	Pointer to the message.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid message pointer. Pointer does not refer to a valid message.

[0095] A message queue coordinates the transfer of messages between threads. A message queue manages messages on a "first-in / first-out" basis. An "owning" thread creates the message queue. After which, external clients or "using" threads may open and put messages onto the queue. Only the "owning" thread should retrieve or pull messages from the queue.

[0096] Message queues are independent of the system processor / communication processor boundary. Simply, "using" threads may open and put messages on both local (on same CPU) and remote messages queues, transparently.

[0097] Each message queue is uniquely identified by a globally visible queue identified ID. A queue identifier that uniquely specifies a single message queue.

Queue IDs span the entire software system: all processors, and all software components.

```
typedef Uint32  Os_Qid_T;
```

[0098] Message priority level (Os_MsgPriority_E) represents the relative priority assigned to a Message. A preferred relative priority is per table 5.

OS_MSG_PRIORITY_LEVEL0	Highest priority level.
OS_MSG_PRIORITY_LEVEL1	.
OS_MSG_PRIORITY_LEVEL2	.
OS_MSG_PRIORITY_LEVEL3	.
OS_MSG_PRIORITY_LEVEL4	.
OS_MSG_PRIORITY_LEVEL5	.
OS_MSG_PRIORITY_LEVEL6	.
OS_MSG_PRIORITY_LEVEL7	Lowest priority level.

Table 5

[0099] The create entry point creates an operating environment message queue based on the requested size, allocating all required underlying OS resources. The caller provides a reference to a handle. This call populates the handle reference with a unique value, identifying the created message queue.

Prototype:

```
Os_Rtn_E
os_createMsgQ( Os_Handle_T   *handle_p,
          Os_Qid_T      qid,
          Uint32        size );
```

Parameters:

*handle_p	Message Queue Handle, filled in by this call on successful creation
Qid	Identifier uniquely specifying this Message Queue.
Size	Size of the requested queue: maximum number of messages.

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_ID_EXISTS	Message Queue for the input QID already exists.
OS_RTN_BAD_PARAM	Bad input parameter detected.
OS_RTN_MEM_ERROR	Failed to allocate required memory.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0100] The destroy entry point destroys or deletes an operating environment message queue, releasing all allocated underlying OS resources. This operation, also, deletes or frees all messages currently queued. After this call, the message queue handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyThread( Os_Handle_T handle );
```

Parameters:

handle	OS Handle, identifying the Message Queue.

Returns:

OS_RTN_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid Message Queue handle.

[0101] The open entry point opens an operating environment message queue. This call binds a "user" thread to a message queue instance, allowing it to transfer messages to the "owning" thread via the queue. The caller provided a reference to a handle. This call will populate this reference with a unique handle, binding the requested Message Queue.

Prototype:

```
Os_Rtn_E
os_openMsgQ( Os_Handle_T   *handle_p,
    Os_Qid_T      qid );
```

Parameters:

*handle_p	Message Queue Handle, filled in by this call on successful creation.
Qid	Identifier uniquely specifying the Message Queue to open or bind.

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_NOT_FOUND	Invalid Queue ID, a Message Queue does not exist for this ID.
OS_RTN_ERROR	General error, failed to bind the Message Queue.

[0102] The close entry point closes an operating environment message queue. This call un-binds a "user" thread and a message queue. After this call, the message queue handle is invalid and is no longer utilized. Any message queue API call performed with a closed handle produces undefined behavior.

Prototype:

```
Os_Rtn_E
os_closeMsgQ( Os_Handle_T   handle );
```

Parameters:

handle	OS Handle, identifying the Message Queue.

Returns:

OS_RTN_OK	Successful close or un-bind.
OS_RTN_BAD_PARAM	Invalid Message Queue handle.

[0103] The put entry point puts a message in an operating environment message queue. The message is placed in the message queue based on its priority. If the "owning" thread is blocked on the message queue, it will wake and receive the message.

If the message is successfully queued, the message queue owns the message until it is "pulled" from the queue.

Prototype:

```
Os_Rtn_E
os_putMsgQ( Os_Handle_T        handle,
        Os_MsgPriority_E   priority,
        void             **msg_pp )
```

Parameters:

handle	OS Handle, identifying the Message Queue.
priority	Priority to assign to the input message. Message placed in the queue based on this priority.
**msg_pp	Reference to a message pointer. If the message is queued successfully, the message pointer is NULLed.

Returns:

OS_RTN_OK	Successful, Message on the tail of the queue.
OS_RTN_BAD_PARAM	Invalid Message Queue handle, or message pointer.
OS_RTN_QUEUE_FULL	Message Queue is full, could not put the message on the queue.

[0104] The put after time entry point puts a message on an operating environment message queue after the indicated time delay. The message is placed at the back or tail of the message queue after the delay period expires. This operation provides the requester with a Cancel ID. The requestor may use the Cancel ID to terminate the put operation up until the delay period expires.

[0105] If this operation successfully sets up the requested delay, the message queue owns the message until it is "pulled" from the queue. If an error occurs "putting" the message after the delay, the message is deleted or freed.

Prototype:

```
Os_Rtn_E
os_putAfterMsgQ( Os_Handle_T        handle,
            Os_TimeMS_T        delay,
            Os_MsgPriority_E   priority,
            void              **msg_pp,
            Os_CancelId_T     *cancelId_p );
```

Parameters:

handle	OS Handle, identifying the Message Queue.
delayMS	Delay in milliseconds. Message will be "Put" after this delay period.
priority	Priority to assign to the input message. Message placed in the queue based on this priority, when the delay expires.
**msg_pp	Reference to a message pointer. If the Put After Time is set-up successfully, the message pointer is NULLed.
*cancelId_p	Cancel ID, returned to the caller. This ID may be used to cancel or terminate this operation.

Returns:

OS_RTN_OK	Successful, Message will be "Put" after the requested delay.
OS_RTN_BAD_PARAM	Invalid Message Queue handle, or message pointer

[0106] The cancel after time entry point cancels or terminates a put after time request. The caller requests the cancel using the Cancel ID. This call deletes or frees the Message tied to the put after time request.

Prototype:

```
Os_Rtn_E
os_cancelPutAfterMsgQ( Os_CancelId_T cancelId );
```

Parameters:

cancelId	Identifier specifying the Put After Time request to cancel or terminate.

Returns:

OS_RTN_OK	Successful, Put After Time terminated, and Message deleted.
OS_RTN_NOT_FOUND	Invalid Cancel ID. A Put After Time request was not found for this ID. This is either an invalid request, or the Put After Time operation has already completed.

[0107] The wait entry point blocks or waits on a message queue until a message appears. If messages are queued when invoked, this call returns the message from the head of the queue immediately. Otherwise, this call blocks the calling thread until a put, or put after time expires. At that time, wait returns the "put" message to the caller. On success, the message ownership is transferred to the caller. The message queue owner, specifically the thread who created the message queue, may only invoke this call.

Prototype:

```
Os_Rtn_E
os_waitMsgQ( Os_Handle_T    handle,
        void        **msg_pp );
```

Parameters:

handle	OS Handle, identifying the Message Queue.
**msg_pp	Reference to a message pointer. Filled in with a Message on success.

Returns:

OS_RTN_OK	Successful, Message pulled from the Message Queue (msg_pp valid).
OS_RTN_BAD_PARAM	Invalid Message Queue handle.

[0108] The get entry point pulls a message from a message queue, if present. If at least one message is queued, this call returns the message from the head of the queue immediately. Otherwise, this call returns queue empty status. On success, the message ownership is transferred to the caller. The message queue owner, specifically the thread who created the message queue, should only invoke this call.

Prototype:

```
Os_Rtn_E
os_getMsgQ ( Os_Handle_T    handle,
        void        **msg_pp );
```

Parameters:

handle	OS Handle, identifying the Message Queue.
**msg_pp	Reference to a message pointer. Filled in with a Message on success.

Returns:

OS_RTN_OK	Successful, Message pulled from the Message Queue (msg_pp valid).
OS_RTN_BAD_PARAM	Invalid Message Queue handle.
OS_RTN_QUEUE_EMPTY	Message Queue is empty.

[0109] A Communication Buffer is a thread synchronization element, allowing a thread to pass a data block or packet to another thread. Communication buffers are explicitly provided to support the movement of large data packets through the system. Communication buffers only contain raw data. Buffers do not contain data types, which reference other buffers or data blocks.

[0110] The allocate entry point allocates a single communication buffer. It returns a void * pointer to the caller, referring to the buffer. The buffer is allocated based on the size parameter. After allocating a buffer, the client may populate the buffer with data.

Prototype:

```
Os_Rtn_E
os_allocCommBuff( Uint32    size,
        void    **buff_pp );
```

Parameters:

Size	Size to allocate for this buffer.
**buff_pp	Pointer to return a void * pointer, referring to the allocated buffer. buff_p points to the first client useable location within the buffer.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid size.
OS_RTN_MEM_ERROR	Insufficient memory available to satisfy this request.

[0111] The free entry point frees or de-allocates a single communication buffer.

Prototype:

```
Os_Rtn_E
os_freeCommBuff( void    *buff_p );
```

Parameters:

*buff_p	Pointer to the buffer.

Returns:

OS_RTN_OK	Successful.
OS_RTN_BAD_PARAM	Invalid buffer pointer. Pointer does not refer to a valid communication buffer.

[0112] Communication FIFOs manage the transfer of data packets (communication buffers) between threads, including transparent data transfer across the processor boundary. This functionality is provided to support software components that must transfer large amounts of data (i.e. UMTS protocol stack PDUs).

[0113] An "owning" thread creates the FIFO. After which, external clients or "using" threads may open and put communication buffers onto the FIFO. Only the "owning" thread should retrieve or pull buffers from the FIFO. FIFOs are independent of the System Processor / Communication Processor boundary. Simply, "using" threads may open and use both local (same CPU) and remote FIFOs, transparently. Each FIFO is uniquely identified by a globally visible FIFO ID.

[0114] The FIFO ID Identifier uniquely specifies a single FIFO. FIFO IDs span the entire software system: all processors, and all software components.

```
typedef Uint32  Os_FifoId_T;
```

[0115] The create entry point creates an operating environment FIFO based on the requested size, allocating all required underlying OS resources.

[0116] The caller provides a reference to a handle. This call populates the handle reference with a unique value, identifying the created FIFO.

Prototype:

```
Os_Rtn_E
os_createFIFO( Os_Handle_T   *handle_p,
        Os_FifoId_T   fifoId,
        Uint32        size );
```

Parameters:

*handle_p	FIFO Handle, filled in by this call on successful creation
fifoId	Identifier uniquely specifying this FIFO.
size	Size of the requested FIFO: number of buffers that the FIFO can manage.

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_BAD_PARAM	Bad input parameter detected.

OS_RTN_MEM_ERROR	Failed to allocate required memory.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0117] The destroy entry point destroys or deletes an operating environment FIFO, releasing all allocated underlying OS resources. This operation, also, deletes or frees all communication buffers currently stored in the FIFO. After this call, the FIFO handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyFIFO( Os_Handle_T   handle )
```

Parameters:

handle	OS Handle, identifying the FIFO.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid FIFO handle.

[0118] The open entry point opens an operating environment FIFO. This call binds a "user" thread to a FIFO instance, allowing it to transfer communication buffers to the "owner" thread via the FIFO. The caller provides a reference to a handle. This call populates this reference with a unique handle, binding the requested FIFO.

Prototype:

```
Os_Rtn_E
os_openFIFO( Os_Handle_T   *handle_p,
         Os_FifoId_T    fifoId );
```

Parameters:

*handle_p	FIFO Handle, filled in by this call on successful creation.
fifoId	Identifier uniquely specifying the FIFO to open or bind.

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_NOT_FOUND	Invalid FIFO ID, a FIFO does not exist for this ID.
OS_RTN_ERROR	General error, failed to bind the FIFO.

[0119] The close entry point closes an operating environment FIFO. This call un-binds a "user" thread and a FIFO. After this call, the FIFO handle is invalid and is no longer utilized. Any FIFO API call performed with a closed handle produces undefined behavior.

Prototype:

```
Os_Rtn_E
os_closeFIFO( Os_Handle_T   handle );
```

Parameters:

handle	OS Handle, identifying the FIFO.

Returns:

OS_RTN_OK	Successful close or un-bind.
OS_RTN_BAD_PARAM	Invalid FIFO handle.

[0120] The put entry point puts a communication buffer into an operating environment FIFO. The buffer is placed at the back or tail of the FIFO. If this operation is successful, the FIFO owns the communication buffer until it is "pulled" from the FIFO.

Prototype:

```
Os_Rtn_E
os_putFIFO( Os_Handle_T   handle,
        void        **buff_pp,
        Uint32        size );
```

Parameters:

handle	OS Handle, identifying the FIFO.
**buff _pp	Reference to a communication buffer. If the buffer is successfully placed on the FIFO, the buffer pointer is NULLed.
size	Size of the buffer in bytes.

Returns:

OS_RTN_OK	Successful, buffer placed at the end or tail of the FIFO.
OS_RTN_MEM_ERROR	Failed to allocate required memory.
OS_RTN_QUEUE_FULL	FIFO is full, could not put the buffer in the FIFO.

[0121] The wait entry point blocks or waits on a FIFO until a communication buffer appears in the FIFO. If the FIFO contains a buffer, this call pulls the buffer and returns the buffer immediately. Otherwise, this call blocks the calling thread until a buffer is "put" in the FIFO. At that time, Wait returns the communication buffer to the caller. On success, the communication buffer ownership is transferred to the caller. The FIFO owner, specifically the thread who created the FIFO, should only invoke this call.

Prototype:

```
Os_Rtn_E
os_waitFIFO( Os_Handle_T   handle,
        void       **buff_pp,
        Uint32       *size_p );
```

Parameters:

handle	OS Handle, identifying the FIFO.
**buf_pp	Reference to a buffer pointer. Filled in with a buffer pointer on success.
*size_p	Buffer size. Filled in with the buffer size on success.

Returns:

OS_RTN_OK	Successful, buffer pulled from the FIFO (buff_pp and size_p valid).
OS_RTN_BAD_PARAM	Invalid FIFO handle.

[0122] The get entry point pulls a communication buffer from a FIFO, if present. Otherwise, this call returns FIFO empty status. On success, the buffer ownership is transferred to the caller. The FIFO owner, specifically the thread who created the FIFO, should only invoke this call.

Prototype:

```
Os_Rtn_E
os_getFIFO( Os_Handle_T    handle,
        void        **buff_pp,
        Uint32        *size_p );
```

Parameters:

handle	OS Handle, identifying the FIFO.
**buff_pp	Reference to a buffer pointer. Filled in with a buffer pointer on success.
*size_p	Buffer size. Filled in with the buffer size on success.

Returns:

OS_RTN_OK	Successful, buffer pulled from the FIFO (buff_pp and size_p valid).
OS_RTN_BAD_PARAM	Invalid FIFO handle.
OS_RTN_QUEUE_EMPTY	FIFO is empty.

[0123] The OS API provides a generic link list implementation with iterator functionality. This feature is provided so that link lists are utilized via a common mechanism throughout the system.

[0124] The create entry point creates an empty list with iterator referring the list end (i.e. NULL element terminating the list). The caller provides a reference to a handle. This call will populate the handle reference with a unique value, identifying the created list.

Prototype:

```
Os_Rtn_E
os_createList(Os_Handle_T *handle_p);
```

Parameters:

*handle_p	List Handle, filled in by this call on successful creation

Returns:

OS_RTN_OK	Successful creation.
OS_RTN_MEM_ERROR	Failed to allocate required memory.
OS_RTN_ERROR	Failed to allocate underlying OS resources.

[0125] The destroy entry point destroys a list and its iterator. The client empties the list prior to destroying it. Otherwise, all elements remaining in the list are leaked, if they are not managed external to the list. After this call, the list handle is invalid and is no longer utilized.

Prototype:

```
Os_Rtn_E
os_destroyList ( Os_Handle_T   handle )
```

Parameters:

handle	OS Handle, identifying the List.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid List handle.

[0126] The append element entry point appends an element to the end of the list with the following special conditions:

- If the list was empty, its iterator will now point to this element at the head.

- If the iterator had advanced past the end of the list, it will now point to the new entry at the tail.

Prototype:

```
Os_Rtn_E
os_appendList( Os_Handle_T  handle,
          void       *item_p);
```

Parameters:

Handle	OS Handle, identifying the List.
*item_p	Element to place in the list, reference via a void pointer.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid List handle.
OS_RTN_MEM_ERROR	Failed to allocate memory required to add the element to the list.

[0127] The insert element before iterator entry point adds an element to the list in front of its iterator, with the following special conditions:

- If the list was empty, the element will be placed at the head, and the iterator will refer to the list end (i.e. NULL element terminating the list).
- If the iterator had advanced past the end of the list, the element will be added at the tail, and the iterator will remain referring to the list end (i.e. NULL element terminating the list).

Prototype:

```
Os_Rtn_E
os_insertBeforeIterator( Os_Handle_T  handle,
                void       *item_p);
```

Parameters:

handle	OS Handle, identifying the List.

*item_p	Element to place in the list, reference via a void pointer.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_BAD_PARAM	Invalid List handle.
OS_RTN_MEM_ERROR	Failed to allocate memory required to add the element to the list.

[0128] The insert element after iterator entry point adds an element to the list in back of its iterator, with the following Special conditions:

- If the list was empty, this request is rejected with OS_RTN_END_OF_LIST status.
- If the iterator had advanced past the end of the list, this request is rejected with OS_RTN_END_OF_LIST status.

Prototype:

```
Os_Rtn_E
os_insertAfterIterator( Os_Handle_T  handle,
                   void       *item_p);
```

Parameters:

handle	OS Handle, identifying the List.
*item_p	Element to place in the list, reference via a void pointer.

Returns:

OS_RTN_OK	Successful deletion.
OS_RTN_END_OF_LIST	Iterator has advanced past the end of the list, cannot insert after it.
OS_RTN_BAD_PARAM	Invalid List handle.
OS_RTN_MEM_ERROR	Failed to allocate memory required to add the element to the list.

[0129] The get element at iterator entry point returns a reference to the element located at the list iterator. The element is not removed from the list, with the following special conditions:

- If the list is empty, this call returns NULL.
- If the iterator has advanced past the end of the list, this call returns NULL.

Prototype:
void*
os_getIteratorItem(Os_Handle_T handle);

Parameters:

handle	OS Handle, identifying the List.

Returns:

NULL	Invalid List, Empty List, or Empty Iterator.
Otherwise	Reference to the element located at the Iterator.

[0130] The remove element at iterator entry point removes the element located at the list iterator. The element is returned to the caller with the following special conditions:

- If the list is empty, this call returns NULL.
- If the iterator has advanced past the end of the list, this call returns NULL.

Prototype:
void*
os_removeIteratorItem(Os_Handle_T handle);

Parameters:

handle	OS Handle, identifying the List.

Returns:

NULL	Invalid List, Empty List, or Empty Iterator.
Otherwise	Element pulled from the Iterator.

[0131] The advanced iterator entry point advances the list iterator one position, with the special conditions:

- The iterator may advance off the end of the list successfully. However once the iterator reaches the end, further attempts to advance it will return: OS_RTN_END_OF_LIST.
- If the list is empty, this call returns OS_RTN_END_OF_LIST.

Prototype:

```
Os_Rtn_E
os_advanceIterator(Os_Handle_T handle);
```

Parameters:

handle	OS Handle, identifying the List.

Returns:

OS_RTN_OK	Iterator advanced to next element.
OS_RTN_END_OF_LIST	Cannot advance the iterator, it's at the end of the list.
OS_RTN_BAD_PARAM	Invalid List handle.
OS_RTN_END_OF_LIST	Iterator is at the end of the list.

[0132] The reset iterator entry point resets the list iterator to the head or front of the list.

Prototype:

```
Os_Rtn_E
os_resetIterator(Os_Handle_T handle);
```

Parameters:

handle	OS Handle, identifying the List.

Returns:

OS_RTN_OK	Iterator advanced to next element.
OS_RTN_BAD_PARAM	Invalid List handle.

[0133] The OS API 120 provides a generic hash table implementation. This feature provides a common hash table mechanism throughout the system. The API 120 preferably supports general dynamic memory allocation from the heap, although others may be used.

[0134] The allocate memory entry point allocates a block of memory from the general heap. Location of the general heap is OS dependent. This function exhibits the same behavior as the Standard IO call: malloc. It utilizes over malloc to support Operating System independence.

Prototype:

```
void*
os_allocMem( Uint32 size );
```

Parameters:

size	Request memory block size.

Returns:

NULL	Memory allocation error.
Otherwise	Reference to the allocated memory.

[0135] The free memory entry point frees or de-allocates a block of memory back to the general heap. Location of the general heap is OS dependent. This function exhibits the same behavior as the Standard IO call: free. It must be utilized over free to support operating system independence. This call is used only with memory references returned from os_allocMem. Other input causes undefined behavior.

Prototype:

```
void
os_freeMem( void *mem_p );
```

Parameters:

*mem_p	Pointer to memory block to free.

Returns:
None.

[0136] Other services not falling into the previous categories are as follows. The delay for milli-seconds entry point blocks the calling thread's execution for the indicated number of milli-seconds.

Prototype:
Os_Rtn_E
os_delayMilliSec(Os_TimeMS_T milliseconds);

Parameters:

milliseconds	Delay period in milli-seconds

Returns:

OS_RTN_OK	Delay period complete.
OS_RTN_ERROR	Could not execute the delay period.

[0137] The get current time entry point returns the current free running D<HRT> time value (in milliseconds).

Prototype:
Os_TimeMS_T
os_getTime(void);

Parameters:
None.

Returns:

Os_TimeMS_T	Current time tick value in milliseconds.

[0138] A high-level design description of the OS abstraction layer is as follows. The OS abstraction layer implements the OS API, providing clients with the operating environment.

[0139] Figure 6 provides a high-level illustration of the OS abstraction layer 114. One of the functions of the OS abstraction layer 114 is to map operating environment entities to the underlying OS, such as a system OS or a communication OS. The OS abstraction layer allows for the operating environment to operate independently of the underlying OS platforms or hardware structure. As a result, code generated for use in one underlying OS/hardware environment can be used in other environments by changes to or by changing the OS abstraction layer. As a result, the resources required to be provided to facilitate an OS/hardware platform change are reduced. Although the preferred OS abstraction layer 114 operates with the operating environment described herein, it can be applied to other environments, such as OSE or VS works.

[0140] The abstraction layer 114 provides the operating environment by implementing the OS API 120. Client software components 222 access the operating environment via the API 120, preferably by using a global IDC_OS_API.h header file.

[0141] The abstraction layer is divided into separate modules, based on OS constructs. Each module provides an implementation for a single construct. Complex construct implementations may utilize lower-level constructs to achieve their functionality.

[0142] These modules are either OS dependent modules 226 or OS independent modules 224. OS dependent modules 226 interface, directly, with the underlying OS 228. A separate implementation appears for each target operating system. OS independent modules 224 do not interface with the underlying OS 228. They either provide all required functionality without the OS, or utilize OS dependent constructs via the OS API 120. Their implementation may be used for any target operating system.

[0143] The mixture of dependent and independent modules 226, 224, utilized to create the OS abstraction layer 114, may be tailored based on the characteristics of each operating system, independently. Specifically, one implementation may use an OS independent module 224, and another may use a dependent module 226. For example, EPOC utilizes an independent message queue module. But, OSE uses a dependent message queue module in order to capitalize on the signaling strengths of OSE.

[0144] The following OS abstraction layer naming convention is preferably utilized to specify which modules are OS independent and which are OS dependent (including to which OS).

[0145] OS Independent Modules:

OS_<construct>.c

Prefix - All independent modules start with the "OS_" prefix.

Construct Name - <construct> field identifies the API construct that this module provides.

[0146] OS Dependent Modules:

<os>_<construct>.c

Prefix - <os> prefix field identifies which Operating System this module applies.

Construct Name - <construct> field identifies the API construct that this module provides.

[0147] The software structure for the OS abstraction layer 114 is relatively simple. The OS abstraction layer 114 contains a module for each OS construct specified in the API 120. These modules may be OS dependent 226 or OS independent 224 based on the characteristics of each target operating system.

[0148] The OS modules fall into one of two categories: simple, and complex or compound. Simple modules are self-contained. They provide all required functionality with little assistance from subordinate components. Complex or compound components either require a complex implementation, or require the use of subordinate components to achieve their desired functionality. These may be OS layer internal components, or standard OS constructs. Internal components are accessed via an OS independent internal interface. They provide common services required by several OS abstraction constructs, but they are not published in the OS API 120. Standard OS constructs utilized as subordinates are accessed via the OS API 120.

[0149] Figure 7 illustrates the OS API software structures. These structures are described in table 6.

Module:	**Description:**	**File Name:**
Simplex 262		
Mutex 239	Implements a Mutex.	<os>_Mutex.*
Event 240	Implements an Event.	<os>_Event.*
Thread 231	Implements a Thread.	<os>_Thread.*
Message 233	Implements the basic Message allocation and de-allocation services.	<os>_Message.*
Generic List 235 with Iterator 237	Implements a generic link list, including a list iterator.	<os>_List.*
Complex/Compound 264		
Message Queue 246	Implements a Message Queue. Message queues provide IPC capabilities, requiring two implementations (or sub-modules): local and remote. A local queue resides on same processor as its client. It includes all message queue capabilities. A remote queue provides a proxy for a queue located on the opposite processor. It provides only "user" capabilities (open, close, and put) via the IPC Management module.	<os>_MsgQ.* <os>_LocalMsgQ.* <os>_RemoteMsgQ.*
FIFO 248	Implements a FIFO. FIFOs provide IPC capabilities, requiring two implementations (or sub-modules): local and remote. A local FIFO resides on same processor as its client. It includes all FIFO capabilities. A remote FIFO provides a proxy for a FIFO located on the opposite processor. It provides only "user" capabilities (open, close, and put) via the IPC Management module.	<os>_Fifo.* <os>_LocalFifo.* <os>_RemoteFifo.*

Module:	Description:	File Name:
Thread Group 242	Implements a Thread Group. A Thread Group manages a collection of Synchronized Threads. This module is OS independent, containing thread management functionality. It utilizes the OS dependent SyncThread module, providing a physical thread implementation within the context of the underlying OS.	<os>_ThreadGroup.*
Memory Management 254	Provides all Memory Management capabilities. Note, this module will be expanded, and may be split into OS dependent and independent components.	<os>_Memory.*
Log / Debug Support 256	Provides the error and debug reporting capabilities. Note, this module will be expanded, and may be split into OS dependent and independent components.	<os>_Log.*
Internal 266		
Time Management 258 and OS Time 260	Provides all time-related functions. This module is a hybrid, providing both API services, and OS Abstraction services. API Services: time now, and delay. Internal Services: Set CallBack, Cancel CallBack. Note, this module is split into OS dependent and independent components.	OS_TimeMgr.c <os>_Time.*
IPC Management 252	Provides all IPC functionality. Design is TBD. This module may be split into common, and OS specific modules.	<os>_IPC.*
Synchronized Thread 244	Implementation of Synchronized Threads. Note, Thread Group's use this internal module exclusively.	<os>_SyncThread.*
Generic Queue 250	Provides a generic queue implementation. Used to create more complex OS constructs.	<os>_Queue.*

Table 6

[0150] The OS constructs, regardless of being OS dependent 226 or independent 224, use the following design pattern. Control blocks store all information required to implement an OS construct within the OS abstraction layer. The control block always begins with a marker specific to the construct type. The marker serves two purposes. First, it is used as a defensive programming measure. Functions that accept a reference to a control block as input always verify the marker before accessing data within the block. Secondly, the markers can be used during debug. They allow control blocks to be easily identified in memory. Typically, control blocks are defined privately within their construct's source file, encapsulating their format and implementation. Control blocks are allocated and freed using dedicated functions. This allows their allocation mechanism to be isolated, and changed without effecting the core construct logic.

[0151] An OS handle is assigned to represent or identify a construct, when the OS layer creates the construct instance. The OS layer uses the address of the construct's control block as the handle value. When a client inputs a construct handle into an OS API call, the OS layer converts the handle into a pointer referencing the control block for that construct instance. There is no need to perform look-ups or searches. This maximizes performance with little cost to size, and makes the OS layer efficient.

[0152] OS constructs provide both a creation method, and a destruction method. The "Create" method allocates and populates the construct's control block, including creating subordinate objects. The "Destroy" method kills all subordinate objects, and frees the construct's control block.

[0153] OS constructs that include IPC capabilities provide open and close methods. These calls bind and unbind clients to construct instances regardless of where they reside in the software system. IPC capable constructs, also, require unique identifiers (Queue IDs, FIFO IDs, etc.).

[0154] As an implication, this feature introduces the concept of owners and users. An "owning" client creates and may destroy a construct instance. The construct allows its "owning" client access to its full feature set. A "using" client binds and unbinds to a single construct instance via open and close. Closing a construct does not destroy it. Additionally, a construct may restrict "using" clients, allowing them access to a sub-set of construct's features. For example, an owner can "wait on" and "put to" a message queue, but a user can only "put to" the queue.

[0155] Figure 8 illustrates the design of the message queue construct. Clients 232 use message queues as per the OS API 120. To implement its required functionality, the message queue 230 utilizes the services of several subordinate modules. The local message queue 234 and the remote message queue 236 represent specialized queue instances. These elements provide the functionality required to perform queue operations locally (i.e. on the same processor as the client) and remotely (i.e. on the opposite processor). The generalized message queue 230 forwards service requests to these subordinates based on queue type.

[0156] A local message queue 234 represents the physical queue, and it is self-contained. This object may be either OS dependent or OS independent. A remote message queue 236 represents a proxy for a queue physically located on the opposite processor. It coordinates the transfer of messages to its associated queue via the IPC Manager 238. The IPC Manager 238 transfers messages across the processor boundary, and utilizes the standard message queue "put" service to queue the message in the physical queue. The IPC Manager contains separate components executing on each processor.

[0157] The object database 240 provides storage and management of active objects (Message Queues and Buffer Queues) within the system. The time manager 242 provides a schedulable callback service. The message queue 230 utilizes this feature to perform "PutAfter", and "CancelPutAfter" requests. This service is provided above (or before) the local/remote distinction. Thus, all "PutAfter" timing executes local to the sending or putting client. This design is used to keep the sending client as close as possible to the timer, coupling the timer to the sender and not the receiver.

[0158] Figure 9 illustrates the message queue control block data structures and their relationships. Both message queue create and look-up operations return a reference to an OS object structure (control block) 245 as the message queue handle. In the local queue case, the object structure refers to a local message queue control block via an OS handle. The local message queue control block 247 contains information required to control a local queue. The message queue module 230 supplies this handle to the local message queue module 234 whenever it forwards service requests. The remote message queue control block 249 contains information required for control of the remote queue.

[0159] For the local message queue, the control block includes: a linked list 251 used to store messages in priority order, an event 253 used to implement the blocking "Wait" message queue call, and a mutex 255 used to protect the generic queue data structure.

[0160] The buffer queue utilizes the same general framework as message queues with the exception of priority. Buffers cannot be placed into a buffer queue based on priority. They are strictly "first-in-first-out".

[0161] The object database provides storage and management of active objects within the system. Objects are defined as OS constructs that are explicitly designed to be shared across thread, and processor boundaries, including message queues which support communication between threads (control and data) and buffer queues which support the transfer of large data blocks between threads.

[0162] Objects are identified using the following information types: Object ID which is a unique object identifier within an application, object type which is the type of the object, and message queue, buffer queue, etc., and application ID which is a unique identifier of each application. The application ID field is used for the FDD SoC requirements. Other information stored for each object is: object handle which is a handle for the actual object and a processor which is a processor location: master / slave.

[0163] Similar to the application ID, the need and use of the processor field is determined as the protocol stack design progresses, specifically in regards to IPC. The distributed environment feature of the OS API is optional.

[0164] The object database 260, as shown in Figure 10, is implemented in an OS-independent manner. Each object has an object structure 258, typically including an object ID, type, application, CPU = Master/Slave and Object reference. The OS client initiates the creation and destruction of these objects. Object references are stored within a hash table 262. A combination of Application ID, Object ID, and Object Type is as the hash table key. The database utilizes a mutex 264 to thread protect the hash table 262.

[0165] The OS layer 114 preferably provides a time management function. The OS layer 114 provides two sets of time-related functionality, API time services and an OS layer internal callback mechanism based on time. The API time services include os_getTime and os_delayMilliSec. The internal callback mechanism allows OS constructs to schedule callbacks based on time (milliseconds). This feature is used to provide their own time related services (for example, the "Put After Time" service provided by Message Queues).

[0166] Time management functionality within the OS layer is divided into two modules as follows:

OS_TimeMgr: contains the Time Manager (which provides callbacks, see below), and all OS independent API calls (os_getTime).

<os>_Time: contains the OS specific Timer used by the Time Manager (this object is typically OS dependent), and all OS dependent API calls (os_delayMilliSec).

[0167] The time manager object 266, as shown in Figure 11, provides most of the time-relative functionality. The API os_getTime entry point is implemented returning the current OS Time (1 ms tick from start-up). It, also, provides the callback service utilized by other OS constructs to provide own time-relative functions. The callback feature allows time manager clients to schedule delayed function calls. They input a function reference, a parameter reference, and a delay. The time manager 266 invokes or "calls back" the function, supplying the indicated parameter, after the delay.

[0168] Callback functions provided to the time manager perform minimal processing in order to maintain timing integrity. Callback execution takes place in the context of the timer thread 270. Extended processing effects other callbacks. As such, complex operations must be handed off to another thread.

[0169] The time manager 266 uses two subordinates to provide this functionality:

- CallBack List 272 is an ordered list of callback requests, earliest (soonest to expire) to latest. This list manages outstanding callbacks, and their information (function, parameter, etc.). The operations of the callback list 272 are to allocate entries 274, delete entries 274, add entries 274, remove entries 274 and to get next expire times.
- Timer 268 provides an internal thread 270 that is woken up or scheduled based on callback delays. Specifically, the time manager 266 sets the timer 268 for the shortest callback delay. When the delay expires, the timer 268 signals the manager 266 to execute the callback. This cycle continues with the time manager 266 always scheduling the shortest delay in the timer 268. When the CallBack List 272 empties, the manager 266 sets the timer 268 for the minimum period required to detect underlying OS tick wrap conditions.

[0170] Figure 12 illustrates a time manager control block functionality. The time manager control block 276 is associated with Muxs 278, 280 and a generic linked list 282. One of the Muxs is the OS system time MUX 278 and the other is a CallBack list Mux 280. The generic linked list 282 (CallBack List) has a time manager list entry 284. The time manager list entry has a Uint64 expire time CallBack function, a CallBack function reference, a CallBack parameter reference and a cancel ID.

[0171] Figure 13 is a preferred flow diagram of a creation/initialization time manager scenario. The OS initiates the bring-up 286 of the time manager, 288. The time manager 266 sends an "os_createList(Os_Handle_T*handle_p)" message to empty the Call Back list, 290, 292. The time manager 266 sends an "os_createMux(Os_Handle_T*handle_p)" message to the system time Mux 278 so that a Mux is produced to protect the OS system time, 294, 296. The time manager 266 sends an "os_createMux(Os_Handle_T*handle_p) to the CallBack list Mux 280 so that a Mux is produced to protect the CallBack list, 298, 300.

[0172] The time manager 266 sends an "os_initTimer()" message to the timer 268 to initialize the time, 302, 303. The timer initialization includes:

- Determining underlying OS tick period.
- Calculating ms / tick.
- Determining the maximum timer value in ms, based on the OS tick period.
- Reading the current OS tick and store it as the "Last Read OS System Tick".
- Creating the timer thread.

[0173] The time manager 266 initializes itself, 304, which includes setting the current OS system time value to 0x0. The manager 266 sends an "os_setTimer(max.timer period) message to the timer 268 so that the indicated timer is set ("Execute CallBacks" called after the indicated delay), 306, 308.

[0174] Figure 14 is a preferred flow diagram of a get OS time time manager scenario. The OS API client 310 send an "os_getTimer()" message to the time manager 266 to update the system time ("updateSystemTime()"), 312, 314. The time manager 266 sends a lock Mux message ("os_lockMutex()") to the system time Mux 278, 316. A "os_getElapsedTime()" message is sent to the timer 268 so that the time elapsed since the last time a call was initiated is determined, 318, 320. The determination includes reading the OS tick, calculating the time based on the current tick, last tick and the tick period, and storing the current tick as the last read tick.

[0175] The time manager 266 updates the current OS system time and adds the elapsed time to the current OS time, 322. A release Mux message ("os_releaseMutex()") is sent to the system time Mux 278, 324. The time manager 266 pulls the low word from the current OS system time to return ("Os_TimeMS_T") and the time is returned to the OS API client 310("return Os_TimeMS_T"), 326, 328.

[0176] Figure 15 is a preferred flow diagram of a set CallBack time manager scenario. The time manager client 330sends an "os_setCallBack" message to the time manager 266, 332. The time manager 266 calculates the CallBack expire time, 334. This calculation includes updating the system time and setting the CallBack time equal to the current system time plus a time delay.

[0177] The time manager 266 sends a loc Mux message ("os_lockMutex()") to the CallBack list Mux 280, 336. The manager 266 sends an allocate CallBack entry message ("Alloc CallBack Entry") to the CallBack list 272, 338. The manager 266 fills in the CallBack Entry with "Expire Time", "callback_fp", "arg_p" and "Cancel ID = running cancel ID++", 340.

[0178] The manager 266 sends an add callback list entry message ("Add CallBack List Entry") to the CallBack list 272, 342. The CallBack list is preferably an ordered list based on the expire time (earliest and latest). An element is added to the list based on its expiration time. Of the entry is added to the head of the list (soonest to first callback entry), the callback timer is canceled and reset based on this delay. The time manager 266 sends a cancel timer message ("Cancel Timer()") to the timer 268, 344. The manager 268 also sends a set timer message ("Set Timer(delay time)") to the timer 268, 346. The delay time is set to the smaller of the time to the next callback or the maximum timer period. The timer 268 sets the indicated timer ("Execute CallBacks" called after indicated delay), 348. The manager 266 sends a release Mux 350 to the CallBack list Mux 280, 350. The cancel ID is returned to the caller, 351. This occurs sometime after the callback delay expires, 352.

[0179] Figure 16 is a preferred flow diagram of a cancel CallBack time manager scenario. The time manager client 330 sends a set CallBack message ("os_setCallBack") to the time manager 266, 354. The time manager 266 schedules a CallBack with the timer active, 356. The time management client 330 sends a cancel CallBack message ("os_cancelCallBack") to the time manager 266, 358. The manager 266 sends a lock Mux message ("os_lockMutex()") to the CallBack list Mutex 280, 360. The manager 266 sends a remove CallBack entry with cancel ID ("Remove CallBack Entry(canceled)" to the CallBack list 272, 362. The Callback list 272 searches the list for an entry with the input cancel ID and if it fins one, it pulls the element from the list, 364.

[0180] The time manager 266 pulls the arg_p from the CallBack entry, 366. The manager deletes the CallBack entry, 368. A release Mux message ("os_releaseMutex ()") is sent to the CallBack list Mux 280, 372and an arg_p is returned to the caller, 370.

[0181] Figure 17 is a preferred flow diagram of an execute CallBack time manager scenario. The processing illustrated in Figure 18 is preferably performed in the timer thread. The timer 268 expires at the CallBack delay, 374. The timer executes a CallBack ("Execute CallBack()") to the time manager 266, 376.

[0182] A loop executes the CallBacks as follows, 378. The manager 266 updates the system timer ("updateSystemTime()"), 380. The manager sends a lock Mux message ("os_lockMutex()") to the CallBack list Mux 280, 382. The manager 266 determines if the CallBack is expired by checking the current time against the next expire time. If the current time is greater than the expire time, the following is performed, 384.

[0183] If the next CallBack expires, 386, a pull CallBack entry message is sent to the CallBack list 272, 388. Callback list Mux 280 is released before the callback function is invoked. It may schedule another callback during its execution (i.e. a callback that performs some processing that includes setting another callback). Other clients are allowed to add their requests during callback execution without blocking. The manager 266 sends a release Mux message ("os_releaseMutex()") to the CallBack list Mux 280, 390. A "callback_fp(arg_p)") message is sent to the time manager client, 392. The manager 266 sends a delete CallBack entry message ("deleteCallBackEntry()") to the CallBack list 272, 394. The procedure loops back to step 380 until all the expired callbacks are completed, 398. The manager 266 sends a set timer message ("os_setTimer(next delay period)") to the timer 268. The timer 268 sets the indicated timer ("Execute Callbacks" called after indicated delay), 400.

[0184] In the preferred embodiment, the timer 268 is replaced with an OS independent time manager thread. This thread is an OS API Thread and encapsulates both the timer (set, cancel, etc), and the "execute callback" functionality. Mapping from the code to the design:

- "pauseCurrentThread" - Set Timer
- "wakeCurrentThread" - Cancel Timer
- "timeMgrMain" - Thread's main loop, includes Timer Thread and execute callbacks.

[0185] Inter-processor communication is primarily performed by an inter-processor message queue interface. The following details the preferred interface between the message queue module and the IPC module (IPC manager and QID database).

[0186] As shown in Figure 18, the message queue 402 has the following IPC elements:

- Message QID Database 408: Used to both register local message queues with the system, and to gain access to remote message queues.
- IPC Manager 406: Remote message queue proxy forwards all "put" requests to its associate queue via the IPC Manager.

[0187] The Message Queue 402 forwards all remote operation to a remote message queue 404. The remote message queue transfers the messages to the IPC manager 406.

[0188] Figure 19 is a flow diagram of message queue creation and destruction in context with inter-processor communication. A first processor has an associated message queue 402_1, message QID database 408_1 and IPC manager 406_1. A second processor also has an associated message queue 402_2, message QID database 408_2 and IPC manager 406_2. The first processor message queue 402_1 receives a create massage, 410. A "qidd_registerMsgQ(qid, handle)" message is sent to the message QID database, 408_1, 412. The database 408_1 searches the local QID database, 414, the remote QID database, 416, and adds the QID to the local QID database with the QID and the handle, 418.

[0189] The queue is now available on the local (first) processor. The message QID database 408_1 sends a "ipc_reportMsgQ(qid, local handle)" to the IPC manager 406_1, 420. The manger 406_1 sends a "IPC cmds to register Q (QID, and handle)" message across the processor boundary to the second processor IPC manager 406_2, 422. The IPC manager 406_2 sends a "qidd_registerRemoteMsgQ(qid, handle in remote context)" to the message QID database 408_2, 424. The queue is now available to the remote (second) processor. The QID is added to the remote database 408_2 with the QID and handle, 426.

[0190] The first processor message queue 402_1 receive a destroy massage, 428. A "qidd_removeMsgQ(qid, handle)" message is sent to the message QID database, 408_1, 430. The database 408_1 searches the local QID database, 432, and removes the QID to the local QID database, 434.

[0191] The message QID database 408_1 sends a "ipc_removeMsgQ(qid)" to the IPC manager 406_1, 436. The manger 406_1 sends a "IPC cmds to remove Q (QID)" message across the processor boundary to the second processor IPC manager 406_2, 438. The IPC manager 406_2 sends a "qidd_removeRemoteMsgQ(qid)" to the Message QID database 408_2, 440. The queue is no longer available on the remote (second) processor. The message QID database 4082 removes the QID from the remote database, 442, and a destroy message proxy ("msgq_DestroyProxy()") is sent to the message queue, 444.

[0192] Figure 20 is a flow diagram of message queue opening in context with inter-processor communication. The message queue 402_1 receives an open message, 446. The message queue 402_1 sends a "qiddb_getMsgQHandle(qid)" message to the message QID database 408_1, 448. The message QID database searches the local QID database for the message registered locally and returns the message queue handle, 450. The local handle is returned.

[0193] At the remote (second processor), if the first remote is open, an open message is received by the message queue 402_2, 452. The message queue 402_2 sends a "qiddb_getMsgQHandle(qid)" to the message QID database 408_2, 454. The message QID database 408_2 searches the local QID database and the remote QID database, 456, 458. If the QID not found in the second processor database, it is created. The QID in the remote database is found, without a proxy handle. The CB is allocated for the proxy: QID, remote handle, and remote type. The proxy now exists. The proxy handle is returned to the caller (open). A create proxy message ("msgq_createProxy(QID, Remote Q_Handle)") is sent to the message Queue 402_2., 462, and a proxy is created, 462. The proxy handle is returned by the message QID database 408_2, 464.

[0194] If all additional remotes are open, the open message is received by the message queue 402_2, 466. The message queue 402_2 sends a get message handle message ("qiddb_getMsgQHandle(qid)") to the message QID database 408_2. The message QID database 408_2 searches the remote QID database, 470. If the QID is found in the local database with a proxy handle, the proxy handle is returned to the caller (open).

[0195] Figure 21 is a flow diagram of message queue remote put in context with inter-processor communication. A put message ("Put(msg_p)") is received by the message queue 402_1, 474. The message queue pulls the handle from the MsgQ CB, 476. The "Type" is set to remote and the "handle" is set to Q handle in the remote processor's context. A put message ("Put(handle, msg_p") is sent to the remote message queue (proxy) 404_1, 478. The remote queue 404_1 sends a "ipc_sendMsg(remote handle, msg_p)" message to the IPC manager 406_1, 480. The IPC manager 406_1 commands the message and handle across the processor boundary to the other IPC manager 406_2, 482. The physical data movement occurs during IPC manager to IPC manger interaction. That IPC manager 406_2 sends a "msgq_putMsgQ(handle, msg_p)" to the message queue 402_2, 484. The message is put in the queue, 486 and processing of the local message queue proceeds.

[0196] The message QID database 488 is illustrated in Figure 22. The message QID database 488 utilizes two lists to manage local 490 and remote queue 492 instances. The local database 490 stores QID and local handle for local queue instances. Similarly, the remote database 492 maintains QID, remote handle, and local proxy handle for remote queues.

[0197] The message queue interface registers the message queue by: verifying the queue does not exist in the system (local or remote), adding the queue to the local message queue database, and informing the remote database of this queue via the IPC manager (IPC report message queue).

[0198] To remove a message queue, the following is performed: deleting the queue from the local message queue database and removing the queue from the remote database via the IPC manager (IPC remove message queue). To get the message queue handle for a local queue, the local queue's handle is returned. For a remote queue, the remote queue's proxy handle is returned. If is does not exist, a proxy creation is triggered via the message queue module (Create Proxy).

[0199] To register a remote message queue, the following is performed: the IPC manager calls when it receives notification that a message queue was created on the opposite processor and the queue is added to the remote message queue database. A proxy is not created until the message queue is opened locally.

[0200] To register a message queue, the following is performed: the IPC Manager calls when it receives notification that a message queue was destroyed on the opposite processor and the queue is removed from the remote message queue database.

* * *